the STRING/80 bit

USER MANUAL

KEYbits
INCORPORATED

P.O. Box 592293
Miami, Florida 33159
(305) 238-3820

COPYRIGHT NOTICE

TRADEMARK

The following names are trademarks of Key Bits Inc.

The STRING Bit
The STRING/80 Bit
STRING/80
Key Bits Inc.
Key Bits Incorporated

DISCLAIMER

Key Bits Inc. makes no representation or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantibility or fitness for any particular purpose. Further, Key Bits Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Key Bits Inc. to notify any person or organization of such revision or changes.

ACKNOWLEDGMENTS

References are made throughout this document to CP/M. Key Bits Inc. would like to acknowledge the fact that CP/M is a trademark of Digital Research of Pacific Grove, California.

the STRING/80 bit

USER MANUAL

April, 1980

Key Bits Inc.
P.O. Box 592293
Miami, Florida 33159

The STRING/80 Bit USER MANUAL

TABLE OF CONTENTS

The STRING/80 Bit USER MANUAL

The STRING/80 Bit USER MANUAL

INTRODUCTION

CHAPTER 1

The STRING/80 Bit USER MANUAL
1 - Introduction

Welcome to the world of Z80/8080 Fortran String Handling

You have just simplified the task of handling character strings in your Z80/8080 computer system environment. The STRING/80 Bit (tm) has been designed for high performance, low burden, character string handling in most Z80/8080 microcomputers using the CP/M or CP/M-Compatible operating system. As a collection, these machine-language routines, developed in assembler language, were specifically designed to interface to Microsoft's Fortran. However, any language processor that conforms to the Microsoft convention of passing call parameters, may obtain direct benefit from The STRING/80 Bit.

The STRING/80 Bit includes the following calls and functions:

Housekeeping Call Routines

VER$	Returns a string with version information and copyright notice.
DEFD$	Initializes a default string variable. This variable is a non-dimensional string space 80 characters in length.
DEFS$	Initializes a string variable of a specified length, up to 255 bytes. Any non-dimensional string variable with a length between 1 and 255 can be initialized using this routine.
DEFT$	Initializes multiple dimension string variables, such as a table or matrix of strings. The entry length of these table or matrix structures can be any length between 1 and 255 characters.
STRIP$	Clears a string of unwanted characters when the string has been used in reading the disk or console.

String Conversion Call Routines

UPPER$	Converts all alphabetic (A-Z) characters in the string to upper case characters regardless of what they were prior to the operation.
LOWER$	Converts all alphabetic character in the string to lower case characters regardless of what they were prior to the operation.
CMD$	Retrieves command parameters from CP/M systems. The default buffer is located at 080 Hex.
MAKE$	Makes a string variable a copy of another string.
MERGE$	Appends a copy of one string to another.
RIGHT$	Places a copy of the rightmost characters of a string into another.
MID$	Fills a string variable with the characters extracted from the middle of another string.
LEFT$	Stores a copy of the extracted leftmost characters of one string into another.
SWAP$	Swaps one string with another.

String Handling Functions

IEOS$	Sets literal end of string marker character to be any character the user desires.
LEN$	Returns length of string.
IDEN$	Compares two strings and returns a -1 if the first is less than the second, a zero if they are equal, and a +1 if the first is larger than the second.

MATCH$	Returns location of one string found within another string.
NVAL$	Returns location of numeric value in string, and moves value to string buffer to allow DECODE conversions to be performed.

File Handling Functions

LOOK$	Looks to see if a given file exists on a disk. The routine is found in the utility library. The Fortran source is included.
NFORM$	Forms a file name into a eleven character string which may be used to open a file.

File Handling Call Routine

CHAIN$	Executes any CP/M program file.
DIR$	Retrieves CP/M directory information. The routine is found in the utility library. Assembler source code is included.
REN$	Renames a CP/M file. The routine is found in the utility library. Fortran source code is included.
KILL$	Kills, or removes a CP/M file. The routine is found in the utility library. Fortran source code is included.
SELCT$	Selects the current CP/M drive assignment. The routine is found in the utility library. Fortran source code is included.
RESET$	Performs a CP/M system reset. The routine is found in the utility library. Fortran source code is included.

System Call and Supporting Function

CPM$	Exercises the CP/M system calls under the control of the Fortran language.
LOC$	Determines the string location and generates a pointer variable that can be passed to CPM$.

Chapter 2 is an overview of the structure of the routines, the conventions employed in their use, and the rules governing their use.

Chapter 3 is an initial discussion of programming using the STRING/80 routines. A simple "list" program is developed to demonstrate the simplicity of the routines and their use.

Chapter 4 is a discussion of the Demonstration Program (DEMO). Although rather simple in function, and admittedly contrived, this demonstration program illustrates at least one use of each of the STRING/80 functions in context of an application.

Included in the demonstration package are several other simple programs that can be viewed using the list program. These include:

DEMO	The host program that calls each of the others and performs several management tasks such as menus in response to HELP, a display of file names in response to DIR, kills a file (KILL), and renames a file (REN).
LIST	A generalized list program as developed in the programing section of the manual.
FORMAL	A program to reformat a Fortran source program and sequentially renumber the labels.
SERIES	A program to name and manage up to three numeric series, each containing up to seventeen entries each. Reporting of the individual series includes the entry count,

	total value, average, minimum and maximum values found.
SORT	A simple sort program capable of sequencing a standard CP/M text file. A list of U.S. presidents in included for the demonstration.

Each of these programs can be run "stand-alone" or as part of the demonstration program.

Chapter 5 is a brief description of some advanced programming techniques associated with The STRING/80 Bit for those who are interested in going further with these routines. This chapter is not intended to be an exhaustive course, but rather a primer for those interested in exploring some of the subtle capabilities inherent in the STRING/80 package.

Chapter 6 is a complete reference section. Each routine is identified and the call parameter sequence is discussed. Unique requirements are identified, and where appropriate, special techniques are reviewed.

The Appendicies contain a quick referecnce quide, identification of the software included on your disk, and a handy user comment and/or problem identification sheet.

The STRING/80 Bit is a powerful enhancement to Fortran in the Z80/8080 envirnonment, especially for those using the CP/M system. We hope you will take your time to become familiar with it, use it, and appreciate it as much as we have.

GENERAL CONVENTIONS AND RULES

CHAPTER 2

Structural Overview

Each STRING/80 Bit function or call routine works upon one or more character strings. The first thing you must do to use the STRING/80 Bit is to define the desired string variables in working storage and give them a name. Once this is done all further references can be accomplished by using the defined name.

Defining a String Variable

Before jumping headlong into the structure of the STRING/80 Bit, let's consider some of the general rules that apply to the use of all strings while in this environment. Perhaps not all of these rules and conventions will be particularly meaningful at this time, however as we proceed the meaning will become more clear. Once you have completed this section, and have examined some of the demonstration (DEMO) code, it is suggested that you return to this section and review these rules and conventions.

Rules

1. All character strings must be defined as BYTE, or character type data.

2. The arrays defined should be large enough to hold the longest string of characters that will be moved into the string variable. This can be an array size of from one to 255 characters. If a string variable of too short a length is inadvertantly used, the STRING/80 Bit truncates the string when out of room.

3. The valid characters in a string include alphabetic (upper and lower case), numerics, all special characters and the TAB character. Another way to state this for those requiring a more precise definition is all characters of hexidecimal values 20H through 7EH, and value 09H. The current keyboard end-of-string (EOS) character can never be used in a string since, by definition, it will terminate the string. The EOS can however be changed using the IEOS$ function to be any desired character. If a zero (binary 0) is assigned using IEOS$, any character may be present in the string.

The default keyboard EOS for the system is an at-sign (@).

4. The maximum character string length allowable is 255 decimal.

5. The character string variable name size can be anything between one and six characters in length. While this provides some flexibility, it is recommended than you try to keep the names as short as possible for sake of code clarity; but not so short that the meaning of the name is lost.

6. Strings used for storing file names must be at least 12 characters in length. An easy way around having to keep track of this is to use a default string (length 80 characters) for this purpose.

7. When compiling and linking Fortran programs, always search the UTL80.REL library (if necessary) before searching the STR80.REL library.

8. The STRING/80 Bit routine CHAIN$ is not suited for storing in ROM memory.

9. Mimic Fortran's use of registers when using the STRING/80 Bit from assembler language.

In addition to the above rules, it is recommended that the following conventions be used. Some of these are presented to make it easier to identify what is happening in your code, for documentation purposes more than anything else. Others are presented to help you avoid some of the situations that can arise. While these are not mandatory, programming will be much easier and simplier if followed, especially when you first start using the STRING/80 Bit.

Conventions

1. String variable names should utilize the '$' as the last position of the name. For example, rather than name a string variable SAM, SAM$ is preferable.

2. Use default strings whenever you can. The only cases not appropriate for the use of default strings is when a string of more than 80 characters is required, storage optimization is a concern, or when a table or matrix of strings is desired.

3. The String variable lengths for storage optimization or tables should be defined as only that needed. If a list of twelve character names is being managed for example, only twelve characters need be defined.

4. Always use default strings to read data from the console or disk and when passing strings to the CP/M operating system.

5. When control strings (defined below) are used, define the control string as 8 bytes in length.

6. Change the end-of-string character (EOS$) as seldom as possible.

7. Do not pass literals (i.e., 1), rather set variables such as N=1 and pass the variable, N, in the parameter list. This MUST be the case for all parameters of the DIR$ and CPM$ statements.

The Type of String Variables

There are two types of character string variables that can be handled in the STRING/80 system, default string variables and control string variables. Further, string variables can be single strings or multiple dimension strings, such as tables of strings or a matrix of strings. A default string can only be a single string of 80 characters and is initialized using the DEFD$ statement. Single strings of lengths other than 80 characters are defined as control strings and are defined using the DEFS$ statement. Tables of strings, whether one dimension or two (matrix), are defined as control strings and are defined using the DEFT$ statement.

In summary, all strings other than the default string are control strings. The Default string is not dimensioned and has a fixed length of 80 characters. The two types of control strings are String, with no dimension, and Table with two dimensions.

The Default String Variable

The default string variable is provided for the common, and most frequent, string variable requirements. The string functions will treat a default string as being 80 characters long. Thus the default string variable takes on the attributes of being a single dimension string variable with a string length of 80 characters.

A default string, with the name A$, is declared as follows:

```
BYTE A$(80)
CALL DEFD$(A$)
```

The dollar sign symbol, while one of the six characters in the name, is used just as a reminder that this is the string name that is passed to the string routines. The dollar sign is not required, any valid Fortran variable name may be used. The benefits of using the "$" symbol will become apparent as we proceed. The second line, or initialization statement DEFD$, define default string, is generally found as one of the first executable statements in the program and is the way the default string variable is initialized. As a result of this action, A$ will appear to be an empty string; or a string of length equal to zero.

While A$ could have been defined as less than 80 characters there would be no protection against "over-running" the length if more than the defined number of characters were presented to it. The user, or programmer, must be aware of and cautious of this limitation. The safest way is to always declare a default string as BYTE with 80 characters.

It should be noted that the default string could have been given more than 80 characters also, however no more than 80 would be used by the routines and thus it would only result in wasted memory.

Controlled String Variables

There are many cases when it is desirable to either define a string variable with a length other than 80 characters or to define an array of string variables, such as a table of names. For these situations, the controlled string variable should be used.

In this case, a control string is declared, preferably with a dollar sign in the name, and the actual data string containing the characters is separately declared, generally without the use of a dollar sign in the name. Consider an array of strings, each string is 32 characters in length and there are 20 strings in total, arranged in a multidimensional array 4 by 5. The DEFT$, or define table variable, is used to define this item. This structure is declared as follows:

```
BYTE A$(8)
BYTE A(32,4,5)
CALL DEFT$(A$,A,32,4,5)
```

or more typically,

```
BYTE A$(8),A(32,4,5)
CALL DEFT$(A$,A,32,4,5)
```

While the three dimensional Fortran array A(32,4,5) actually contains the two-dimensional string array characters, the eight byte array A$ is the array name that is used in conjunction with each string routine. Thus the actual Fortran code, whether referencing a default string variable or a controlled string variable, would be similar.

The DEFT$ routine initializes the multi-dimensional array and links it to the "control" string. It declares a 3 dimension array, with 32, 4 and 5 being the sizes of each dimension, under the name A, to be controlled by the control string A$ and hereafter referenced by the name A$.

Granted, any other name could have be chosen (Sam and Carl for example), but for code clarity and improved documentation A$ and A were selected. In accordance with the recommended convention, names such as JOHN$ and JOHN or NAME$ and NAME will do as adequate a job.

Here is another example of a control string variable declaration, this time a 50 entry table of twenty characters each:

```
BYTE NAME$(8),NAME(20,50,1)
CALL DEFT$(NAME$,NAME,20,50,1)
```

Included in this control string class for string variables is the non dimensional, non 80 character type string variable. This type of string is defined using the DEFS$ statement (define string). In a simple example where we want to define a string to hold a file name, the definition would look like this:

```
BYTE FILE$(8),FILE(12)
CALL DEFS$(FILE$,FILE,12)
```

Strictly speaking, the length of the control string need be only 4 bytes for a string and 8 bytes for up to a two dimension array. It is recommended that a convention of declaring the control string as an 8 byte string always be followed. Please note, that in all cases except where you specifically choose to do byte manipulation, the control string (A$) is used and not the data string (A).

Now that we know how to define each of the three

essential string types (Default, String and Table), let's look at the rest of the statements and their syntax.

The STR80 and UTL80 Command Syntax

Throughout this document we will use the following command syntax:

```
CALL NAME$(A$,[N1,N2],B$,[N1,N2],[N3],[N4])
```

where;
- NAME$ - is the statement name,
- A$,B$ - are string names using the convention of a $ in the last position of the name as a reminder that this is the name used when referencing string variables.
- N1,N2, N3,N4 - are integer value parameters

and;
- [] - indicates the optional items.

In the above example, note that when N1 is required so is N2, while N3 or N4 are independent options.

The STRING/80 Bit allows almost all integer parameters in the parameter lists to be literals, i.e., the routines do not change their values. The DIR$ and CPM$ routines are exceptions to this since they do change the values passed. However, it is still recommended that all parameters be specified as variables.

The following is a list of each routine and their syntax. Remember that while the string library allows all types of string usage, the utility library requires that default strings only be passed. These default strings are shown as S$.

Housekeeping Call Routines

```
CALL VER$(A$,[N1,N2])

CALL DEFD$(A$)

CALL DEFS$(A$,B$,N1)

CALL DEFT$(A$,B$,N1,N2,N3)

CALL STRIP$(A$,[N1,N2])
```

String Conversion Call Routines

```
CALL UPPER$(A$,[N1,N2],B$,[N1,N2])
CALL LOWER$(A$,[N1,N2],B$,[N1,N2])
CALL CMD$(A$,[N1,N2])
CALL MAKE$(A$,[N1,N2],B$,[N1,N2])
CALL MERGE$(A$,[N1,N2],B$,[N1,N2])
CALL MID$(A$,[N1,N2],B$,[N1,N2],N3,N4)
CALL RIGHT$(A$,[N1,N2],B$,[N1,N2],N3)
CALL LEFT$(A$,[N1,N2],B$,[N1,N2],N3)
CALL SWAP$(A$,[N1,N2],B$,[N1,N2])
```

String Handling Functions

```
I=IEOS$(A$)
I=LEN$(A$,[N1],[N2])
I=IDEN$(A$,[N1,N2],B$,[N1,N2])
I=MATCH$(A$,[N1,N2],B$,[N1,N2],N3)
I=NVAL$(A$,[N1,N2],B$,[N1,N2],N3)
```

File Handling Functions

```
I=LOOK$(S$)                          (UTL80)
I=NFORM$(A$,B$,N1)
```

File Handling Call Routines

```
CALL CHAIN$(A$,[N1,N2])

CALL DIR$(S$,N1)                    (UTL80)

CALL REN$(S$)                       (UTL80)

CALL KILL$(S$)                      (UTL80)

CALL SELCT$(N1)                     (UTL80)

CALL RESET$                         (UTL80)
```

System Call and Supporting Function

```
CALL CPM$(A,BC,DE,HL)               (passed parameters are
                                     integer values)
I=LOC$(A$,[N1,N2])
```

In the next chapter we will begin programming with the STRING/80 Bit using the statement definitions presented in this chapter. Before proceeding, it is recommended that you review some of the rules and conventions discussed at the beginning of Chapter 2.

THIS PAGE INTENTIONALLY LEFT BLANK

PROGRAMMING WITH THE STRING/80 BIT

CHAPTER 3

Getting Started

In this chapter we will develop an application using the STRING/80 bit collection of tools. We will walk you through each step of the program developement to allow you to gain an insight into some of the capabilities and features. When completed you should have a software product that will serve you in many different capacities in the future.

Beginning a programming project in any environment requires a definition of what you would like the program to do, a functional specification. Our project, selected for the purpose of demonstrating some of the programming techniques used when developing a program using the STRING/80 bit, will be a LIST program. This will be a program that, when executed, will list the contents of any text file on the screen, similar to the TYPE command in the CP/M system. Since you will already have this capability, you are no doubt asking yourself, "Why do I need another?". The answer is simple. You would like a list program with some additional features not found in the typical LIST or TYPE program.

First, let's look at the basic functional requirements. The most essential part of a list program is to be able to execute it simply. The primary design issue is to have the program respond to the following command:

LIST d:nnnnnnnn.xxx

where;

d - is the drive identity, A, B, C or D. No specified drive will default to the currently assigned drive.

nnnnnnnn - is the file name, and

xxx - is the file extension.

If only LIST is specified the program will respond with an operator prompt such as the following:

Enter file name to be listed -

In either case, whether a file was specified or entered as a result to the operator prompt, the program checks to see if the file exists prior to opening it, and if it does not

exist, prints the following message at the terminal:

*** FILE NOT FOUND ***

In this situation, the program returns with the previously mentioned operator prompt. The user can now enter a valid file name or abort the program by entering either the word END or a control-C (C). Note however that a control-C returns control directly to the operating system, while END returns control to the LIST program. The program can now do an orderly abort and return control to the operating system or to some other program as can be seen in our next feature.

It is desirable to have the ability to run the LIST program from another program and to have the LIST program return control to a specified program. We will develop the program to return control to the DEMO program for obvious reasons.

That should be enough to keep it simple while at the same time provide additional functions not found in the common list program. Now let's develop the program to meet these objectives.

Program Organization

As an aid to the upcomming coding task an outline, or organization flow of the program, should first be considered. The organization of our case program is as follows:

Housekeeping	This code names the program, declares any data variables and constants required by the program and then initializes them.
Program Identification	Any startup message or bulletin is printed with this code.
Retrieve Command Line	The command line or parameters entered are retrieved from the operating system.
Validate File Name	This code checks to see that the file name retrieved from the command line is a valid file name for an existing file on the disk.

Format File Name	The now valid file name is formatted for use in a Fortran OPEN statement.
Open File	Open the file we're going to list.
Change String Delimiter	Just before we start reading the text file, the end-of-string character is changed to be a non-text character to prevent it's presence in the file from inadvertantly interfering with our task at hand.
Read	Read and format the input string.
Write	Write the string out to the terminal.
Prompt Operator	If an invalid file name was found in the Validate File Name routine, control is passed to this routine to prompt and retieve a new file name.
Read Error Message	If an error is encountered during a disk read, control is passed to this routine to print an error message and try reading the disk again.
End of Program	If during a read the end-of-file is encountered, control is passed to this routine for an orderly termination of the program. Any sign off messages are displayed here. If END is encountered as a file name after an operator prompt, control is passed to this routine rather than look for another file name.
Program Return Check	This routine, executed after the end of program routine, checks to see if control should be passed to another program or back to the operating system.
Chain to Program	If control should be passed to another program, the Program Return Check routine passes control to here.

End	If control should be passed to the operating system, the Program Return Check routine passes control here to perform that function.

Now let's examine the code for each one of these routines individually. Once we have discussed each one separately, we'll look at the whole program in context.

Housekeeping

As in any Fortran program, the program is given a name and the necessary data variables are declared for subsequent use. In our case the program will be called LIST and the source code will be put in a file LIST.FOR. The standard editor is all that is needed to prepare the code.

Data variables required for this program are very few. Three string variables are used to process the various file names and disk records. FILE$ was chosen as our name for the files string variable. It is defined as 80 characters in length as a Default String Variable. BUFF$ was chosen as the string variable for buffering the disk records prior to printing them at the terminal. Although this could have also be defined as a default string variable, for purposes of demonstration a Controlled String Variable was used. All string variables were defined as BYTE arrays.

The default variable is initialized by DEFD$. Initialization for the control string is accomplished via the DEFS$ routine. Notice that for control strings, the initialization routine is the vehicle that associates the control string BUFF$ with the byte array BUFF. Once initialized all references to BUFF$, FILE$, or NAMES$ can be treated equivalently with the one exception of the read statment which we look at a little later.

The initial code looks like the following:

```
        PROGRAM LIST

C---------------------------------
C  DEFINE STRING VARIABLES
C---------------------------------

        BYTE FILE$(80)
        BYTE BUFF$(8), BUFF(80)
        BYTE NAME$(8), NAME(12)

C---------------------------------
C  INITIALIZE STRING VARIABLES
C---------------------------------

        CALL DEFD$(FILE$)
        CALL DEFS$(BUFF$,BUFF,80)
        CALL DEFS$(NAME$,NAME,12)
        LEOS=0
```

Program Identification

It is generally a good approach for the program to identify itself to the user. In our case, we not only want to identify the program name and version information, but we also want to communicate the version number of the STRING/80 bit being used in the program. An example of the type of message to be displayed is as follows:

```
The LIST Bit (tm) - Version 1.01
Copyright (C) 1980 Key Bits Inc.
January, 1980 - Miami, Fl, USA

            using

The STRING/80 Bit (tm) Version 1.16
```

The code to support this includes several write statements, the FILE$ variable, and the VER$ call. The VER$ statement is to retrieve the version identification.

```
C---------------------------------
C  PRINT PROGRAM IDENTIFICATION
C---------------------------------

        WRITE(5,10)
10      FORMAT(1X,12X,1X)
        WRITE(5,20)
20      FORMAT(1X,12X,'The LIST Bit (tm) - Version 1.02')
        WRITE(5,30)
30      FORMAT(1X,12X,'Copyright (C) 1980  Key Bits Inc.')
        WRITE(5,40)
40      FORMAT(1X,12X,'January, 1980  -  Miami, FL, USA')
        WRITE(5,10)
        WRITE(5,50)
50      FORMAT(1X,25X,'using')
        WRITE(5,10)
        CALL VER$(FILE$)
        J=LEN$(FILE$)
        WRITE(5,60)(FILE$(I),I=1,J)
60      FORMAT(1X,80A1)
        WRITE(5,10)
        WRITE(5,10)
```

The Command Line Parameter Retrieval

When the user enters the CP/M command to execute the program, it is possible that he entered a string of characters such as the following:

```
LIST TEST.FOR
```

The CP/M system uses the characters 'LIST' to execute this program. It converts the string to all upper case characters and then leaves the residual of the command line, i.e. ' TEST.FOR', in the system for the programs use. Note that there is a leading blank in the remaining string. This string of characters could of course be anything, including nothing, and are the parameters being passed to the program. This data, stored at 080 Hex in the CP/M system, will possibly be destroyed during the execution of our program. As such, it is desirable to retrieve and save this information as early in the program as possible. The CMD$ call does this task.

```
 C--------------------------------
 C  RETRIEVE PARAMETERS
 C--------------------------------

         CALL CMD$(FILE$)
         I2=MATCH$(FILE$,'#@',1)
         J=LEN$(FILE$)
         IF(J.LT.2)GO TO 90    (Prompt Operator for File Name Routines)
```

In this case the FILE$ string was once again used as the string variable in which to save our parameter information. Further, since there is at least one leading blank, we can conclude that our parameter string must be at least 2 characters in length to be valid (blank plus 1 character). If it is less than 2 characters we can conclude that no file name parameter is present and thus go to the Prompt-Operator-For-File-Name routine.

Check for Valid File Name

If any data is found in the parameter string it is assumed to be a file name and must be validated. This is true whether we get the parameter string data from CMD$ or by prompting the operator to enter it (which we'll look at next). The simpliest way to validate the file name is to look on the disk(s) and see if the file of that name exists. The LOOK$ function is used to do this:

```
 C--------------------------------
 C  VALIDATE FILE NAME
 C--------------------------------

 70      CONTINUE
         I=LOOK$(FILE$)
         IF(I.GT.-1) GO TO 120    (Format File Name Routine)
         WRITE(5,80)
 80      FORMAT(1X,'*** FILE NOT FOUND ***')
         GO TO 90    (Prompt Operator for File Name Routine)
```

As you can see from the code, the variable 'I' will contain a -1 if no valid file name is found or some positive value if it is. In this case then, if 'I' is ever greater than a minus one we have found a valid file name and can go to the Format File Name Routine. If not, we want to write a

"File Not Found" message to the operator and go to the Prompt Operator for File Name Routine.

Prompt Operator for File Name

If no vaild file name is found, the operator gets a prompt and the response is placed in the same variable, FILE$, as was used to store the results of the previous CMD$ call.

```
C-----------------------------------
C  PROMPT OPERATOR FOR NAME
C-----------------------------------

90      CONTINUE
        WRITE(5,100)
100     FORMAT(1X,'Enter file name to be listed - ')
        READ(5,110)FILE$
110     FORMAT(80A1)
        CALL STRIP$(FILE$)
        CALL UPPER$(FILE$)
        J=LEN$(FILE$)
        IF(J.NE.3)GO TO 70
        I=MATCH$(FILE$,'END@',1)
        IF(I.GT.0)GO TO 180   (End of Program Routine)
        GO TO 70   (Validate File Name)
```

The STRIP$ statement marks the end of the string. Fortran reads from the disk or console as if they were card readers with 80 positions. This means there will be trailing blanks. STRIP$ removes the trailing blanks and marks the end of the string.

There is one other thing we need do in this routine. If the operator responds with the word 'END' or 'end', we want to terminate the program. After converting the FILE$ variable data to all upper case characters with the UPPER$ statement, we scan the data for a match with the MATCH$ statement. If a match is found the variable 'I' will contain a value representing the location of the word 'END' in the variable FILE$. If no match is found, 'I' will contain a zero.

If the variable 'I' is greater than zero control is passed to the End of Program Routine. If it is zero, the

Check for Valid File Name Routine is executed. Notice how the length was tested to see if it was only three characters before we looked for 'END'. This is to avoid interpreting a file name such as 'SEND.TXT' erroneously.

Special attention should be given to the literal 'END@' in the MATCH$ statement. The at-sign within the quotes is the last character of any string constant. This character can be changed or set to a different chracter by using the IEOS$ function. Note however, that all string constants MUST be terminated by the CURRENT VALID end-of-string character.

Format File Name, Open File, Change EOS

The vaild file name needs to be formatted for acceptance by Fortran. The Microsoft compiler expects a numeric disk drive value (0=current, 1=A, 2=B, 3=C, 4=D) and eleven characters for the file name. For example, if the operator entered "B:TEST.FOR" as the parameter string, it need be converted into a variable, such as 'K', and a string variable, such as FILE$, wherein the new string value looks like "TEST   FOR". The NFORM$ statement does that job for us.

```
C---------------------------------
C  OPEN VALIDATED FILE NAME
C---------------------------------

120     CONTINUE
        K=NFORM$(NAME$,FILE$,1)
        CALL OPEN(7,NAME,K)
        LEOS=IEOS$(0)
```

The NAME$ variable now has the formatted file name and 'K' has the numeric drive number. The standard Fortran OPEN statement can now be immediately used to open the file. Note however that the OPEN statement uses the data string NAME and not the control string NAME$.

Everything is now ready to proceed, except for one thing. If the list program were ever to read a copy of itself or read a copy of a file with an end-of-string (EOS) marker in it, typically an at-sign, the end-of-string character would terminate the line on which the EOS were found and thus produce invalid or truncated results.

In order to avoid this, we can simply reassign the value of the end-of-string chracter (EOS) with the IEOS$ statement. In this case we reassigned it a numeric value zero (a binary 0), a character that would never show up in a string of characters in a text file. Now we are ready to read the first record.

Read a Record From the File

The character string variable that we decided to use for our buffer when reading the disk file is BUFF$. As you recall, BUFF$ is a control string that is linked to the real data string BUFF via the INIT$ statement. While we could have defined BUFF$ to be a default string, for purposes of illustration we are using it in the control-string configuration. Note, that had we used a default string, the standard Fortran READ statement would have used BUFF$. Since BUFF$ is a control string, the READ statement (and the WRITE statement) must use the real data string BUFF. This is one of those rare occasions when the actual string need be referenced. It also highlights the value of establishing consistent variable naming conventions (BUFF$ versus BUFF).

In either case, whether a control string or default string, the READ statement uses the byte read format, i.e. 80A1.

```
C--------------------------------
C  READ A RECORD FROM FILE
C--------------------------------

130     CONTINUE
        READ(7,140,ERR=160,END=180)BUFF
140     FORMAT(80A1)
        CALL STRIP$(BUFF$)
```

Write a Record to the Screen

The implicit DO-loop is used with the real data array to write the control string variable to the screen or to the disk. In order to set the size of the string and in turn the limit of the implicit DO-loop, 'J' is set to the length of the string, or buffer, BUFF$, immediately prior to writing it out.

```
C----------------------------------
C WRITE A RECORD TO SCREEN
C----------------------------------

        J=LEN$(BUFF$)
        IF(J.LE.0).OR.(J.GE.80)GO TO 130
        WRITE(5,150)(BUFF(I),I=1,J)
150     FORMAT(1X,80A1)
        GO TO 130   (Read a Record from the Disk)
```

Print a Read-Error Message

In case a read error is encountered a message will be printed to the operator by this routine thus letting him know of the event. Control will then be passed to the read routine to read again. No special string handling was used in this routine. It's all standard Fortran. Note however that we could have used MAKE$ to make a string variable such as MESS$, for message, contain the message and then print MESS$ using the write-routine. Had we, it would have looked like, "CALL MAKE$(MESS$,'*** READ ERROR ***@')". However we didn't, so it looks like this:

```
C----------------------------------
C  PRINT READ ERROR MESSAGE
C----------------------------------

160     CONTINUE
        WRITE(5,170)
170     FORMAT(1X,'*** READ ERROR ***')
        GO TO 130   (Read a Record from the Disk Routine)
```

End of Program Routine

When the READ statement encounters the end-of-file it passes control to this routine. The first thing we do is reset the end-of-string character back to its default value, '@' in case we want to use it in our program (which we do). We then print a 'Goodbye' message.

```
C-----------------------------------
C  END OF PROGRAM ROUTINE
C-----------------------------------

180     CONTINUE
        LEOS=IEOS$('@')
        WRITE(5,10)
        WRITE(5,190)
190     FORMAT(1X,12X, '        Goodbye . . .')
        WRITE(5,10)
        WRITE(5,10)
```

Check for Return to a Program

At this point we have the option to return to a calling program or the operating system. Since we will use this program as part of the DEMO package, for example, the same program can be used stand-alone or used as part of the DEMO. In our case the DEMO program called the LIST program and placed a pound-sign (#) as the last character of the parameter string, after a blank, and after all the data parameters. Thus a DEMO initited LIST would look like:

```
LIST TEST.FOR #
```

An operator initiated LIST would look the same but without the last blank and '#'. We can now test if a '#' was present, the flag I2 was set in the Retrieve Parameter routine to reflect its presence. If present a "chain" back to the DEMO program will be made. If not present, control will go to the operating system via the normal END statement.

```
C-----------------------------------
C  CHECK FOR RETURN TO A PROGRAM
C-----------------------------------

        IF(I2.GT.0)GO TO 200   (Chain to Calling Program)
        GO TO 220   (Return to Operating System)
```

In the next routine we need to use the standard end-of-string. It is IMPORTANT that the EOS be reset back to its default after we complete reading the disk file.

Chain to the Calling Program

The CHAIN$ statement is used to call another program. The string name is the file name of the program followed by the end-of-string character, i.e. 'DEMO.COM@'. Notice, that if the CHAIN$ fails, control falls through to an error message. This could happen if the DEMO.COM file were not found on the current disk, for example.

```
C----------------------------------
C  CHAIN TO THE CALLING PROGRAM
C----------------------------------

200     CONTINUE
        CALL CHAIN$('DEMO.COM@')
        WRITE(5,210)
210     FORMAT(1X,'*** DEMO.COM NOT FOUND - RETURNED TO SYSTEM ***')
```

Return to Operating System

To return to the operating system, the standard Fortran END statement is executed.

```
C----------------------------------
C  RETURN TO OPERATING SYSTEM
C----------------------------------

220     CONTINUE
        END
```

Summary

You have now completed your first program using the STRING/80 Bit. Many of the features were used. Those that were not essentially work the same way. The best way to see this for yourself is to begin using them in you own programs, referring to the REFERENCE MANUAL chapter (Chapter 6) and the other demonstration programs for examples. Once oriented, these tools will vastly improve your productivity in Fortran and keep your string handling effort to a minimum.

The Whole LIST Program

```
        PROGRAM LIST

C--------------------------------
C  DEFINE STRING VARIABLES
C--------------------------------

        BYTE FILE$(80)
        BYTE BUFF$(8), BUFF(80)
        BYTE NAME$(8), NAME(12)

C--------------------------------
C  INITIALIZE STRING VARIABLES
C--------------------------------

        CALL DEFD$(FILE$)
        CALL DEFS$(BUFF$,BUFF,80)
        CALL DEFS$(NAME$,NAME,12)
        LEOS=0

C--------------------------------
C  PRINT PROGRAM IDENTIFICATION
C--------------------------------

        WRITE(5,10)
10      FORMAT(1X,12X,1X)
        WRITE(5,20)
20      FORMAT(1X,12X,'The LIST Bit (tm) - Version 1.02')
        WRITE(5,30)
30      FORMAT(1X,12X,'Copyright (C) 1980  Key Bits Inc.')
        WRITE(5,40)
40      FORMAT(1X,12X,'January, 1980  -  Miami, FL, USA')
        WRITE(5,10)
        WRITE(5,50)
50      FORMAT(1X,25X,'using')
        WRITE(5,10)
        CALL VER$(FILE$)
        J=LEN$(FILE$)
        WRITE(5,60)(FILE$(I),I=1,J)
60      FORMAT(1X,80A1)
        WRITE(5,10)
        WRITE(5,10)

C--------------------------------
C  RETRIEVE PARAMETERS
C--------------------------------

        CALL CMD$(FILE$)
        I2=MATCH$(FILE$,'#@',1)
        J=LEN$(FILE$)
        IF(J.LT.2)GO TO 90
```

```
C---------------------------------
C  VALIDATE FILE NAME
C---------------------------------

70      CONTINUE
        I=LOOK$(FILE$)
        IF(I.GT.-1) GO TO 120
        WRITE(5,80)
80      FORMAT(1X,'*** FILE NOT FOUND ***')
        GO TO 90

C---------------------------------
C  PROMPT OPERATOR FOR NAME
C---------------------------------

90      CONTINUE
        WRITE(5,100)
100     FORMAT(1X,'Enter file name to be listed - ')
        READ(5,110)FILE$
110     FORMAT(80A1)
        CALL STRIP$(FILE$)
        CALL UPPER$(FILE$)
        J=LEN$(FILE$)
        IF(J.NE.3)GO TO 70
        I=MATCH$(FILE$,'END@',1)
        IF(I.GT.0)GO TO 180
        GO TO 70

C---------------------------------
C  OPEN VALIDATED FILE NAME
C---------------------------------

120     CONTINUE
        K=NFORM$(NAME$,FILE$,1)
        CALL OPEN(7,NAME$,K)
        LEOS=IEOS$(0)

C---------------------------------
C  READ A RECORD FROM FILE
C---------------------------------

130     CONTINUE
        READ(7,140,ERR=160,END=180)BUFF
140     FORMAT(80A1)
        CALL STRIP$(BUFF$)

C---------------------------------
C WRITE A RECORD TO SCREEN
C---------------------------------

        J=LEN$(BUFF$)
        IF((J.LE.0).OR.(J.GE.80)GO TO 130
```

```
        WRITE(5,150)(BUFF(I),I=1,J)
150     FORMAT(1X,80A1)
        GO TO 130


C-----------------------------------
C  PRINT READ ERROR MESSAGE
C-----------------------------------

160     CONTINUE
        WRITE(5,170)
170     FORMAT(1X,'*** READ ERROR ***')
        GO TO 130

C-----------------------------------
C  END OF PROGRAM ROUTINE
C-----------------------------------

180     CONTINUE
        LEOS=IEOS$('@')
        WRITE(5,10)
        WRITE(5,190)
190     FORMAT(1X,12X, '        Goodbye . . .')
        WRITE(5,10)
        WRITE(5,10)

C---------------------------------
C  CHECK FOR RETURN TO A PROGRAM
C---------------------------------

        IF(I2.GT.0)GO TO 200
        GO TO 220

C---------------------------------
C  CHAIN TO THE CALLING PROGRAM
C---------------------------------

200     CONTINUE
        CALL CHAIN$('DEMO.COM@')
        WRITE(5,210)
210     FORMAT(1X,'*** DEMO.COM NOT FOUND - RETURNED TO SYSTEM ***@')

C---------------------------------
C  RETURN TO OPERATING SYSTEM
C---------------------------------

220     CONTINUE
        END
```

THIS PAGE INTENTIONALLY LEFT BLANK

THE DEMONSTRATION PROGRAMS

CHAPTER 4

The Demonstration Program Functions

The Demonstration Programs have been provided as examples of how various STRING/80 commands can be used in real case situations. All of the demonstration programs can be run from the program DEMO.COM. You will note that each is in fact a separate program in the directory. They execute each other by using the CHAIN$ facility found in the STRING/80 Bit. In addition, each can be run separately.

The collection of demonstration programs include the following:

DEMO	A control program to run the following programs in a predictable and controlled fashion.
LIST	A program to list any file and then return control to the DEMO program. This is the same program that was developed in Chapter 3.
SERIES	A program to name up to three series of 17 elements each, load, update and maintain the data within the series, and calculate the total, average, count, minimum, maximum of each series. Upon termination control is returned to the DEMO program.
FORMAL	A program to reformat and resequence the numeric labels of a Fortran source program then return control to the program DEMO.
SORT	A program to sort a list of names and then return control to the program DEMO. The demonstration file used is the PRES.TXT, a list of U.S. President's names in the sequence of their term in office.

In addition to the collection of programs listed above, there are several functions contained within the main DEMO program worth noting. These illustrate some of the STRING/80 commands and in particular some of the CPM$ functions.

HELP	This command displays a description of each function or command supported and a summary menu of all functions.
DIRectory	The currently assigned disk directory is displayed on the console, in bytes rather than Kbytes. (Record Count times 128 bytes)
RENname	The file of your choice is renamed.
KILL	The file of your choice is deleted from the disk directory.

The demonstration programs are very much self explanatory with the possible exception of the program SERIES. Before providing you with a little more explanation on SERIES, please remember that the most effective way to get the feel for how each program works and its capabilities is to try them out.

The SERIES program uses principles that can be put to use in an expanded version to make a very useful time series analysis and reporting system. What follows is a simple senario of the included SERIES program.

```
.
.
.
COMMAND? 1=FIRST                          FIRST is the name assigned
                                          to series number 1.

1=FIRST                                   The command line is echoed.

SERIES NAMES                              The series names are list-
------------                              ed to confirm action.
1 - FIRST
2 - NO NAME
3 - NO NAME

COMMAND? FIRST(6)=1,2,3,4,5,6,7,8,9,8.5   The series FIRST is given
                                          numeric values starting in
                                          the 6th position.
```

```
COMMAND? REPORT FIRST                    Display series FIRST.

         SERIES FIRST
  1               0.0000
  2               0.0000
  3               0.0000
  4               0.0000
  5               0.0000
  6               1.0000
  7               2.0000
  8               3.0000
  9               4.0000
 10               5.0000
 11               6.0000
 12               7.0000
 13               8.0000
 14               9.0000
 15               8.5000
 16               0.0000
 17               0.0000
 TOTAL           53.5000
 AVERAGE          5.3500
 COUNT           10.0000
 MINIMUM          1.0000
 MAXIMUM          9.0000

 COMMAND?
 .
 .
 .
```

As a reminder, these demonstration programs are included as illustrative aids for you to follow in your use of the STRING/80 Bit and not necessarily here as examples of superb proramming techniques. In all cases, these demonstration programs were contrived as a platform upon which to model the STRING/80 Bit routines.

SOME ADVANCED PROGRAMMING TECHNIQUES

CHAPTER 5

The Extended Capabilities of the STRING/80 Bit

The potential capabilities of the STRING/80 Bit go beyond the obvious and are available to those who wish to perform extended string handling tasks. The objective in this chapter is to introduce the adventurous user to some of the architectural subtleties of the STRING/80 Bit. Through several carefully selected examples we will illustrate how these architectural features can be put to use. After that it will be up to the user to explore these advanced techniques using his own ingenuity.

Essentially, the material in this chapter takes advantage of and builds upon the basic STRING/80 Bit routines, using both Fortan and assembler languages, through the knowledge and use of some of the inherent idiosyncrasies of the STRING/80 Bit design. This material will be discussed in the following order:

1. The Control String Definition
2. Manipulation of the Control String Byte(s)
3. CP/M Calls from the Fortran Language
4. Assemble Language use of the STRING/80 Bit routines.

Some basic working knowledge of CP/M and assembler, although not mandatory, is assumed on behalf of the user for the remainder of this chapter. Those finding themselves uncomfortable with the material may wish to review Microsoft's Fortran documentation on the handling of registers and Digital Research's documentation on the CP/M architectrue.

The Control String

Up to now, the control string has been defined as an eight byte string for any control string created. This is not necessarily always the case. The eight bytes represent the maximum number of bytes required. In fact, the required length of the control string varies with the number of dimensions desired in the string array, as follows:

String Dimensions	Bytes Required	
none	4	
1	6	(not used)
2	8	

This will perhaps be more obvious after taking a close look at the structure of the control string. When fully utilized, i.e. in a two dimension array, the control string byte allocation is as follows:

Byte	Purpose
1	String Type Designator
2	Data String Address - Low Byte
3	Data String Address - High Byte
4	String Length
5	Row Address - Low Byte
6	Row Address - High Byte
7	Column Address - Low Byte
8	Column Address - High Byte

All types of strings use byte 1, default strings and control strings. Default strings use byte one as a storage location of something other than a binary one, two, or three. All control strings use bytes 1,2,3,4.

Double dimensioned (control) strings, or tables (single and double dimensioned) use bytes 1 through 8.

Let's examine each of these bytes in more detail and look at how they are used by each of the STRING/80 Bit routines.

Control String Byte 1 - The Data String Type Designator

All STRING/80 routines utilize this byte to determine the type of processing that must be performed upon the character string being referenced. The possible values and their meanings are as follows:

Byte 1 Value	String Type Referenced
1	A non-dimensional character string located at the address specified in bytes 2 and 3, with a length of that specified in byte 4, and referenced by this control string. This is why a non-dimensional character string requires a control string of four bytes.
2	Not Used
3	A double dimensioned character string (such as a matrix of names) whose first entry or string is located at the address specified in bytes 2 and 3, with each of the string entries having a fixed length of that specified in byte 4, and with the number of horizontal, or row entries being that specified in bytes 5 and 6, and with the number of vertical or column entries being that specified in bytes 7 and 8. This is why a double dimensioned character string requires a control string of eight bytes.
0,4-255	Default String.

From the definition of the first byte of the control string it should now be clear, at least preliminarily, as to the purpose and use of bytes 2 through 8. Let's look at each of them a little more closely.

Control String Bytes 2 and 3 - The Data String Address

The initialization routines (DEFS$,DEFT$) associate a control string (example, BUFF$) with a data string (example, BUFF) by storing the physical memory address of the data string in bytes 2 and 3 of the control string. Byte 2 is the low address byte and byte 3 is the high address byte.

As an example, if the data string (BUFF) were stored by the Fortran compiler at the physical memory address 0184 Hex (388 Decimal, 0000 0001 1000 0100 Binary), the value stored in the second and third bytes would be 132 and 1, respectfully.

Control String Byte 4 - The Data String Length

The data string length is stored in byte 4 at the time of initialization by the appropriate routine. Since only eight bits are available in a single byte, the maximum string length is limited to 255 Decimal (0FF Hex, 1111 1111 Binary).

Control String Bytes 5 and 6 - The Row Count

The row count is stored in bytes 5 and 6 at the time of initilization by the appropriate routine. These two bytes allow up to 65,535 decimal strings to be stored in a single table; more than there is memory to accomodate.

Control String Bytes 7 and 8 - The Column Count

The column count is stored in bytes 7 and 8 at the time of initialization by the appropriate routine. These two bytes allow more columns than memory will permit to be stored in a double dimensioned string.

Using the Control String Knowledge

Once aware of the conrol string structure and byte allocation it is easy to see how this knowledge can be put to use. First, let's look at the typical (and recommended) method of defining and initializing a single dimensioned string variable (table).

```
.
.
.
BYTE TABLE$ (8), TABLE (13,600,1)
CALL DEFT$ (TABLE$,TABLE,13,600,1)
.
.
.
```

In this example we have defined a table of 600 entries, each having 13 characters. The physical location of TABLE in memory is however a function of the compiler.

Let us assume, for purposes of illustration, that we did not want to depend on the compiler to locate the table, but rather, we always want the table (TABLE) to start at location E000 Hex (57344 Decimal). Implementing such a design is simple and straightforward.

The byte array TABLE is NOT declared and DEFT$ is NOT used for the task of initialization. Instead, simple assignment statements and the STRING/80 Bit routine MAKE$ is used as follows:

```
          .
          .
          .
          BYTE TABLE$(8)              (eight bytes needed)
          TABLE$(1)=3                 (table dimensioned string)
          TABLE$(2)=0                 (location - low byte - 00)
          TABLE$(3)=224               (location - high byte - E0)
          TABLE$(4)=13                (entry string length)
          TABLE$(5)=88                (row count - low byte 88)
          TABLE$(6)=2                 (row count - high byte 512)
          TABLE$(7)=1                 (column count - low byte 1)
          TABLE$(8)=0                 (column count - high byte 0)
          DO 10 I=1,600
          CALL MAKE$(TABLE$,I,1,'@')      (null string)
   10     CONTINUE
          .
          .
          .
```

Once this has been done, TABLE$ can be referenced by Fortran and STRING/80 routines exactly as if it had been defined by the compiler in the more traditional and recommended method.

Using the logic of this example, we can find many uses for this technique. As illustrated, high memory was used to store a table. Another example might be that of passing data between programs without having to go to disk. The data can now be passed via a TABLE that is physically located "above" the highest position of the largest program in the participating group (remember Fortran places your program stack at the top of user RAM). Another case is that of referencing a specific location of a program already in memory. And, with a little imagination, you can surely think of many more.

CP/M Calls from the Fortran Language

Two special routines have been included in the STRING/80 Bit to allow full use of the standard CP/M calls from the Microsoft Fortran language. These routines are:

CPM$ — This routine is used to load the registers and pass the parameters to the CP/M system.

LOC$ — This routine is used to locate the string address and create a pointer to the address that can be passed using the CPM$ routine.

Perhaps the easiest way to view the operation of these two routines is to look at the source code of KILL$. KILL$ is written using Fortran, as are REN$, RESET$, SELCT$ and LOOK$, and utilizes the CP/M call(s) to accomplish it's function. In the case of KILL$, the CP/M command is number 19 or 13Hex. The file control block (FCB) address is passed in the DE-Register pair. The following is the total code for the KILL$ subroutine.

```
SUBROUTINE KILL$(A$)
INTEGER A,BC,DE,HL, DRV
BYTE FCB$(35)
BYTE A$(1), B$(32)
EQUIVALENCE (B$(1),FCB$(2))

CALL DEFD(FCB$)
CALL DEFD(B$)
DRV=NFORM$(B$,A$,1)
DE=LOC$(FCB$)
FCB$(1)=DRV
BC=19
CALL CPM$(A,BC,DE,HL)
RETURN
END
```

Perhaps the only unusual technique is that of defining the same string space (FCB$) as both FCB$ and B$ with the equivalent statement. Everything else is straight forward and to the point. This is done so NFORM$ places the formatted file name in position 2 of the file control block.

Assembler Language use of the STRING/80 Bit Routines

As mentioned earlier, the STRING Bit routines can be used by assembler, or any language that conforms to the Microsoft Fortran convention of parameter passing. In the DIR$ routine included in the utility library, and as used in the DEMO program, the STRING/80 Bit routine NFORM$ is used to format the file control block.

The source code below is a partial listing of the DIR$ routine, only that portion dealing with the use of the routine NFORM$. For a complete look at the DIR$ routine source code use LIST to display a copy of UTL80.ASM.

```
        .
        .
        .
        ENTRY IDIR$,DIR$
;
        EXT NFORM$
;
; IDIR$ is called from fortran by a call with one default
; string passed as a parameter. This means that the HL-Reg
; contains a pointer to the string address. We would like
; to use the I=NFORM(A$,B$,1) function to format the passed
; string to our file control block (FCB). We can do this by
; passing pointers to the NFORM$ function in a fortran like
; manner.
;
IDIR$:  XRA     A       ;ZERO A REGISTER
        LXI     D,FCB+1 ;THIS IS THE POINTER TO
                        ;THE RECEIVING STRING (A$)
        XCHG            ;PLACE PASSED POINTER IN SECOND POSITION,
                        ;THE DE-REG, AND THE FIRST IN THE HL-REG.
        LXI     B,VALP  ;THIS POINTS TO THE PASSED THIRD
                        ;PARAMETER, THE PASSED INTEGER.
        MOV     M,A     ;PLACE A ZERO (NULL DEFAULT STRING) MARKER
                        ;IN OUR RECEIVING STRING (ie. A$).
;
; We now look like a posible fortran like call. Remember the
; HL-Reg will return our function value, the drive number in
; our case.
;
        CALL    NFORM$  ;FILL OUR FCB WITH THE FILE NAME
        MOV     A,L     ;GET THE LOW VALUE BITS, THEY ARE ENOUGH
        STA     FCB     ;PLACE THE DRIVE REQUEST VALUE
;
; The FCB name is now formed, we must leave a search instruction
; for the first call. DIR$ later sets this to a search next comman
;
```

```
        MVI     A,17    ;LOAD A SEARCH DIRECTORY COMMAND
        STA     COMMD   ;SAVE FOR LATER USE
        RET
        .
        .
        .
```

This concludes the discussion on some ways in which you can use the STRING/80 Bit routines to do more than the obvious. From here on it is up to you to be creative and reap the rewards accordingly. Best of luck; and remember, don't be afraid to experiment; you'll enjoy the results for years to come.

THIS PAGE INTENTIONALLY LEFT BLANK

THE REFERENCE MANUAL

CHAPTER 6

Housekeeping Call Routines

VERSION (VER) This routine retrieves the current version and copyright information on the STRING/80 Bit routines, and returns it in string A$.

```
CALL VER$(A$,[N1,N2])

A$=string
N1=first dimension
N2=second dimension
```

DEFINE DEFAULT STRING (DEFD) This routine initializes the named default string to a non-dimensional 80 character fixed length string variable of length zero. (Places binary zero in first position)

```
CALL DEFD$(A$)

A$=data array variable
```

DEFINE STRING (DEFS) This routine initializes a non-default, non-dimensional string variable.

```
CALL DEFS$(A$,B$,N1)

A$=control string
B$=data array
N1=length in bytes
```

DEFINE TABLE (DEFT) This routine initializes tables of strings of one or more dimensions.

```
CALL DEFT$(A$,B$,N1,N2,N3)

A$=control string
B$=data array
N1=length in bytes
N2=first dimension
N3=second dimension
```

STRIP STRING This routine strips nonrelevant characters from a string that has been read from the console or disk. All trailing blanks are suppressed.

```
CALL STRIP$(A$,[N1,N2])

A$=the string to be stripped
N1=first dimension
N2=second dimension
```

String Conversion Call Routines

UPPER This routine converts all characters in the string to upper case character, regardless of what they were prior to execution.

```
CALL UPPER$(A$,[N1,N2],B$,[N1,N2])

A$=first string, the string to be returned
B$=second string, the string to be converted
N1=first dimension
N2=second dimension
```

LOWER This routine converts all characters in the string to lower case characters, regardless of what they were prior to execution.

```
CALL LOWER$(A$,[N1,N2],B$,[N1,N2])

A$=first string, the string to be retruned
B$=second string, the string to be converted
N1=first dimension
N2=second dimension
```

COMMAND (CMD) This routine retrieves a copy of the CP/M 'command' line from the operating system and puts it in A$.

```
CALL CMD$(A$,[N1,N2])

A$=string in which to store command line
N1=first dimension
N2=second dimension
```

MAKE This routine makes the character string A$ equal to the character string B$.

```
CALL MAKE$(A$,[N1,N2],B$,[N1,N2])

A$=first string, the string returned
B$=second string, the string passed
N1=first dimension
N2=second dimension
```

MERGE This routine merges, or concatenates, the character string B$ to the end of the character string A$.

```
CALL MERGE$(A$,[N1,N2],B$,[N1,N2])

A$=first string, the new string
B$=second string, the string to be added
N1=first dimension
N2=second dimension
```

RIGHT This routine makes the character string A$ equal to the right N3 most number of characters of character string B$.

```
CALL RIGHT$(A$,[N1,N2],B$,[N1,N2],N3)

A$=first string, the new string
B$=second string, the string passed
N1=first dimension
N2=second dimension
N3=number of characters
```

MIDDLE (MID) This routine makes the character string A$ equal to the portion of B$ that starts in position N3 and continues for N4 characters.

```
CALL MID$(A$,[N1,N2],B$,[N1,N2],N3,N4)

A$=first string, the new string
B$=second string, the string passed
N1=first dimension
N2=second dimension
N3=starting character
N4=number of characters
```

LEFT This routine makes character string A$ equal to the left N3 most number of characters of the character string B$.

```
CALL LEFT$(A$,[N1,N2],B$,[N1,N2],N3)
```

A$=first string, the new string
B$=second string, the string passed
N1=first dimension
N2=second dimension
N3=number of characters

SWAP This routine swaps string A$ for string B$ and visa versa.

```
CALL SWAP$(A$,[N1,N2],B$,[N1,N2])
```

A$=first string
B$=second string
N1=first dimension
N2=second dimension

String Handling Functions

INITIALIZE END OF STRING (EOS) This routine sets the end-of-string (EOS) character to any desired character or symbol. The only exclusions are that the EOS cannot be either a binary 1, 2, or 3. Any other binary byte between 0 and 255 decimal is allowed. The system keyboard default is an at-sign (@ a 040Hex, a 64 decimal). The value N1 is the end-of-string character desired and I1 is the value of the end-of-string character immediately prior to being reset to N1.

```
I1=IEOS$(N1)
```

N1=integer, set value
I1=integer, last value
(no string delimiter necessary).

LENGTH (LEN) This routine finds the length of character string A$. Note that TAB characters will be counted as ONE (1) character if found in the string (not 8) as will legitimate blanks and non-printable characters, such as the bell-character. I1 will contain the length upon return.

```
I1=LEN$(A$,[N1,N2])

A$=string
N1=first dimension
N2=second dimension
I1=returned integer length
```

IDENTICAL (IDEN) This routine compares two string. A -1 is returned if A$<B$, a 0 if equal, and a +1 if A$>B$.

```
I1=IDEN$(A$,[N1,N2],B$,[N1,N2])

A$=first string
B$=second string
N1=first dimension
N2=second dimension
I1=returned integer value
```

MATCH This routine finds the first occurance of the string B$ in character string A$, starting in character position N3, and returns a zero (0) in I1 if not found, or returns the starting position of B$ (within A$) in I1, if found.

```
I1=MATCH$(A$,[N1,N2],B$,[N1,N2],N3)

A$=first string
B$=second string
N1=first dimension
N2=second dimension
N3=starting character
I1=returned integer position for start of match
```

NUMERIC VALUE (NVAL) This routine finds the first numeric value occuring after position N3 in B$ and stores it in A$. A decimal point is placed in position 20. The numeric value is centered around position 20 of A$. A$ must be at least 40 bytes long (a default string of length 80 is recommended). This function is intended to work in conjunction with DECODE. A$ can then be used with the FORTRAN function DECODE to convert characters to numerics. Upon return from the routine, I1 contains the character position immediately FOLLOWING the numeric value found in B$. The FORTRAN function ENCODE may be used to convert numerics to characters and place them in a byte array.

```
I1=NVAL$(A$,[N1,N2],B$,[N1,N2],N3)

A$=first string, stored number as characters
B$=second string, the string to be scanned
N1=first dimension
N2=second dimension
N3=starting character
I1=returned integer value, next position
```

File Handling Functions

LOOK This routine looks for the presence of a file with the name as stored in the character string S$. If not found, a minus one (-1) is returned in I1, if found a zero (0) or greater is returned to I1. The source for this routine is included in the utility library. The number returned is the directory entry number if it exists (0 to 63).

```
I1=LOOK$(S$)

S$=string, file name
I1=returned integer value
```

NAME FORM (NFORM) This routine formats a string into a legitimate file name for the Fortran language in the CP/M environment.

```
I1=NFORM$(A$,B$,N1)

A$=first string, formated name returned
B$=second string, unformated name passed
N1=integer, file name position in string B$
I1=integer, drive number
```

File Handling Call Routines

CHAIN This routine passes control to the program name specified in the string variable A$. The string A$ should look like a command string as typed from the console.

```
CALL CHAIN$(A$,[N1,N2])

A$=first string, file name
N1=first dimension
N2=second dimension
```

DIRECTORY (DIR) These routines provide the capability to retrieve directory information. The IDIR$ and DIR$ routines work together. The IDIR$ routine MUST be called first to intialize the retrieval sequence. The CP/M call rules apply. There must be no intervening disk I/O between subsequent DIR$ calls. A directory list for example would begin with the IDIR$ routine being passed a 12 character string ????????.???, and the DIR$ would then be called to retrieve each entry and its identification. The string S$ as recommended, should be a default string. If another type is selected, it should be at least 12 characters in length.

```
CALL IDIR$(S$)

S$=file search (mask) string  ('D:AAAAAAAA.BBB')


CALL DIR$(S$,N1)

S$=string returned
N1=integer, returned record values
        (MUST be a variable)
```

RENAME (REN) This routine renames a disk file. The string S$ is a string containing the command line as normally found in the CP/M environment,i.e. NEW.TXT=OLD.TXT. This routine uses the CALL CPM$ routine. The source code for this routine is included in the utility library.

```
CALL REN$(S$)

S$=assignment string
```

KILL This routine deletes, or kills, a disk file with the name found in the character string S$. This routine used the CALL CPM$ routine. The source code for this routine is included in the utility library.

```
CALL KILL$(S$)

S$=assignment string
```

SELECT DRIVE (SELCT) This routine selects the drive specified in N1.

```
CALL SELCT$(N1)

N1=drive number
```

RESET This routine does a system reset to reinitialize the disk directory maps similiar to what takes place when a CTL-C is entered from the console. This routine used the CALL CPM$ routine. The source code for this routine is included in the utlility library.

```
CALL RESET$
```

System Call and Supporting Function

CPM This routine allows the FORTRAN user to utilize the standard calls in the CP/M Operating System by following the conventions specified in the Digital Research CP/M INTERFACE MANUAL. Specific examples of how this command is used can be found in Chapter 5, Some Advanced Programming Techniques.

```
CALL CPM$(P1,P2,P3,P4)

P1=value placed in the A-register
P2=value placed in the BC-register
P3=value placed in the DE-register
P4=value placed in the HL-register
```

LOCATION (LOC) This routine identifies the location of a string and converts it to a pointer to be used by the CPM$ routine.

```
I1=LOC$(A$,[N1,N2])

A$=string
N1=first dimension
N2=second dimension
I1=pointer value
```

APPENDIX

Software Included On The STRING/80 Bit Distribution Disk

STR80.DOC	The STRING/80 Bit distribution disk documentation file listing the contents.
STR80.REL	The STRING/80 Bit Library
UTL80.FOR	The FORTRAN source code for selected library routines, namely the CALL CPM enhancement package.
UTL80.ASM	The Assembler source code for the DIR$ routine. This routine is included as part of the utility library.
UTL80.REL	The Utility Library.
DEMO.FOR	The FORTRAN source code for the demonstration monitor program.
DEMO.COM	The executable demonstration monitor program.
FORMAL.FOR	The FORTRAN source code for the reformat and resequence program. This program is part of the demonstration package.
FORMAL.COM	The executable reformat and resequence program. This program is part of the demonstration package.
LIST.FOR	The FORTRAN source code for the file list program. This program is part of the demonstration package.
LIST.COM	The executable file list program. This program is part of the demonstrations package.
SERIES.FOR	The FORTRAN source code for the series reporting program. This program is part of the demonstration package.
SERIES.COM	The executable series reporting program. This program is part of the demonstration package.
SORT.FOR	The FORTRAN source code for a simple sort program. This program reads standard CP/M text files and sequences it by individual line. This program is part of the demonstration package and uses file PRES.TXT

for demonstration purposes.

SORT.COM — The executable sort program. This program is part of the demonstration package.

PRES.TXT — This is a text file listing the U.S. presidents in order of their term in office. This file is used by SORT in the demonstration package and will be converted to an alphabetical list of presidents names.

External Labels Referenced by The STRING/80 Bit Modules

The following is a list of external entry point names that are referenced by some of The STRING/80 Bit library modules. The names should be cautiously allocated as routine names that will be used in conjunction with The STRING/80 Bit package.

CHAIN$ CMD$ CPM$ DCODE$ DEFD$ DEFS$ DEFT$ DET$ DIR$ EOS1$
GET$ IEOS$ IDEN$ ITRACE$ KILL$ LEFT$ LEN$ LEN1$ LOC$ LOOK$
LOWER$ MAKE$ MATCH$ MERGE$ MID$ MID1$ M001$ M002$ M003$
M004$ M005$ NFORM$ NVAL$ POS50$ PUT$ REN$ RESET$ RIGHT$
RITE2$ RITE3$ R001$ R002$ R003$ R004$ R005$ R006$ R007$
R008$ SELCT$ STRIP$ SWAP$ T001$ T002$ T003$ T004$ T005$
UPPER$ VER$

The STRING/80 BIT Command Quick Reference Guide

Housekeeping Call Routines

```
        CALL VER$(A$,[N1,N2])
        CALL DEFD$(A$)
        CALL DEFS$(A$,B$,N1)
        CALL DEFT$(A$,B$,N1,N2,N3)
        CALL STRIP$(A$,[N1,N2])
```

String Conversion Call Routines

```
        CALL UPPER$(A$,[N1,N2],B$,[N1,N2])
        CALL LOWER$(A$,[N1,N2],B$,[N1,N2])
        CALL CMD$(A$,[N1,N2])
        CALL MAKE$(A$,[N1,N2],B$,[N1,N2])
        CALL MERGE$(A$,[N1,N2],B$,[N1,N2])
        CALL RIGHT$(A$,[N1,N2],B$,[N1,N2],N3)
        CALL MID$(A$,[N1,N2],B$,[N1,N2],N3,N4)
        CALL LEFT$(A$,[N1,N2],B$,[N1,N2],N3)
        CALL SWAP$(A$,[N1,N2],B$,[N1,N2])
```

String Handling Functions

```
        I=IEOS$(P1)
        I=LEN$(A$,[N1,N2])
        I=IDEN$(A$,[N1,N2],B$,[N1,N2])
        I=MATCH$(A$,[N1,N2],B$,[N1,N2],N3)
        I=NVAL$(A$,[N1,N2],B$,[N1,N2],N3)
```

File Handling Functions

```
#       I=LOOK$(S$)
        I=NFORM$(A$,B$,N1)
```

File Handling Call Routines

```
        CALL CHAIN$(A$,[N1,N2])
+       CALL DIR$(S$,N1)
#       CALL REN$(S$)
#       CALL KILL$(S$)
#       CALL SELCT$(N1)
#       CALL RESET$
```

System Call and Supporting Function

```
CALL CPM$(A,BC,DE,HL)
I=LOC$(A$,[N1,N2])
```

\# - Fortran Source Code Included
\+ - Assembler Source Code Included

A$,B$	are string names using the convention of a $ in the last position of the name as a reminder that this is the name used when referencing string variables.
N1,N2,N3,N4	are numeric integers (1,2,3....n) that are generally dimension subscripts of an array (of strings). In several cases these represent action codes, string termination values, position values etc.
P1	are either A$-type string variable names or N1-type integer variable names as defined above.
S$	is a uniquely defined character string variable consisting of multiple optional parameters. This type of string is used to exchange file parameter information, such as the file assignment data as follows: NEWNAME.TXT=OLDNAME.TXT/L.
A,BC,DE,HL	are integer variables representing the values of the respective registers being passed to CP/M.

The Software Cross Reference Guide

	LIST	FORMAL	SERIES	SORT	DEMO	FILE routines
VER$	X	X	X	X	X	
DEFD$	X	X	X	X	X	
DEFS$	X		X			
DEFT$		X	X	X		
STRIP$	X	X	X	X	X	
UPPER$	X	X		X		
LOWER$						
CMD$	X	X	X	X		
MAKE$		X	X	X	X	
MERGE$		X			X	
RIGHT$		X	X	X	X	
MID$		X			X	
LEFT$		X	X		X	
SWAP$				X		
IEOS$	X	X				X
LEN$	X	X	X	X	X	
IDEN$			X	X		
MATCH$	X	X	X	X	X	X
NVAL$		X	X			
LOOK$	X	X		X		X
NFORM$	X	X		X	X	
CHAIN$	X	X	X	X	X	
DIR$					X	
REN$		X			X	
KILL$		X			X	
SELCT$						
RESET$						
CPM$						X
LOC$						X

User Comments and/or Problem Reporting Form

The last page of this manual (The BUG Bit) is a convenient form upon which you may place you comments or criticisms. Should this form be missing, please feel free to send us your comments anyway, to:

KEY BITS INC.
P. O. BOX 592293
MIAMI, FL 33159

I N D E X